Sunday, 27th April 2008

At this point, it's been almost six weeks since the dissertation was last worked on, as the time was spent revising for and sitting exams. It probably seems slightly mad to back working on this after only one day off (and one night out!), but I'm afraid of losing the momentum that I've built up over the exam period. It's this momentum that I need to keep going for the next five weeks.

I spent today trawling Google Finance to gather a selection of accounts which will be used for evaluating the software, and training it. It's an uninspiring task, which is why I'm doing it now rather than closer to the testing phase. Tomorrow, I plan to continue working on the GUI with an aim to getting it finished by the end of the day.

One thing that has become apparent is that different companies have their accounts prepared in different currencies. Therefore, it's going to be important to work with *ratios* rather than any direct values. But, seeing as the music will be generated from a comparative analysis of financial statements, this shouldn't be too much of a problem.

Objectives Achieved: Some sample accounts collected.

Monday, 28th April 2008

Today I tackled the GUI interface. It allows the following to be done:

- Input 5 parameters for the accounts of two separate years
- Choose an approach algorithm (mapping, genome, etc.)

The usability can be worked on if there's time at the end of the dissertation. Additionally, we can already read in accounts as CSV files.

Objectives Achieved: Functional GUI implemented.

Tuesday, 29th April, 2008

First of all, I think it would be helpful for me to reflect on the feedback from the second deliverable. Overall, I took the following points on board:

- Implementation is very behind schedule.
- Implementation needs to be generated quickly, so that there is sufficient time for evaluation.
- The ideas need to be explained in more specific terms, ideally with examples which demonstrate the processes involved.
- Some examples of sample accounts and expected sample output need to be shown.
- A discussion of specific algorithms rather than methodologies needs to be included.

It seems I neglected to report the half of the specifics which were in my head. And the other half? These will be discovered by performing the implementation, so in a way the design will be finalised by implementation; another good reason why much of it should have been done when submitting the design document.

That said, I'll begin to make up for it now, by going through the day's work.

Firstly, we now have a finished importer. As described in the deliverable, it takes two CSV files for account information. Here are some more specifics (including today's

modifications):

- It's a Java class called 'AccountReader'.
- It currently expects two CSV files, but will be modified shortly to allow input from the GUI.
- It returns an object consisting of three arrays: One contains the column headings for both files, and the other two contain the values.
- It performs several checks on the incoming data:
    - That the headings of both files are identical.
    - That both files have the same number of values as each other.
    - That both files have the same number of values as they do headings.
- It is currently set up for reading in an account sheet.
- The account sheet should have the following parameters in the following order: Current Assets, Total Assets, Current Liabilities, Total Liabilities, Total Equity.
- It has safeguards to check that the account sheet balances correctly (eg: that the Total Assets is equal to the sum of Total Liabilities plus the Total Equity)

So, I give you an example input (Sony's account sheet from Google Finances), which will be used to explain the whole Mapping implementation process.

Last Year's Accounts (file 1)

Current Assets	Total Assets	Current Liabilities	Total Liabilities	Total Equity
4546723	11716362	3551852	8345658	3370704

This Year's Accounts (file 2)

Current Assets	Total Assets	Current Liabilities	Total Liabilities	Total Equity
3769524	10607753	3200228	7403901	3203852

Of course, as they're in CSV format, file 1 looks more like this:

```
Current Assets,Total Assets,Current Liabilities,Total Liabilities,Total Equity
4546723,11716362,3551852,8345658,3370704
```

(notice that the commas make tokenising the files nice and easy ;)

Next up, is the beginning of the Mapping implementation. This is done in Jython (aka: Python with Java compatibility). The Mapping class is made up purely of functions (no Object Orientated design is used). I'll first of all go through the functions as they currently stand:

- prepareAccounts(): This takes in an 'AccountReader' with all its unpleasant arrays, and returns a nice list of pairs for each parameter of the account. Therefore, the above accounts would be pared off as [(4546723, 3769524), (11716362, 10607753), (3551852, 3200228), ...]. This makes it nice and easy to compare changes in the account.
- generateSignals(): If we look at the design for the Mapping approach, we see that each pair of parameters needs to produce a signal. The way I did this was to take a ratio of last year's to this year's. So, the signal for Total Assets would be a float: (11716362 / 10607753). Same for the rest. A higher ratio indicates a more favourable change.
    - But! in cases of liabilities, an increase **is bad**! Therefore, we have a mask which means that these parameters are reversed to generate the ratio. That way, all signal values > 1 are good, and all signal values < 1 are bad.

- Next up, we have the so-called 'signal processors'. These are the boxes which take a raw signal, and turn it into something musical. Today, I have so far implemented the following:
    - signalProcessorTempo(): Takes in a signal, and produces a tempo in BPM. How does it do this? Well, it's essentially a function which maps a signal (ratio) to a tempo value within certain bounds.
    - signalProcessorKey(): Takes in a signal, and is used to generate a key signature. This is done by mapping the signal to a value from 0 - 11.
- the getKeyName() function takes in a value generated by signalProcessorKey() and returns the note of the key signature (eg: gS means the key is in g sharp).
- signalCombinator() takes in a list of signals, and outputs a single signal based on the mean.

So, we can now generate signals from accounts, and even process some of them into musical attributes. I should point out that many of the values can be tweaked with ease in the code. This will be useful when I'm listening to the musical output and checking how it sounds.

I will give pseudocode for the algorithms used here in the final dissertation. However, in the mean time I have been careful to comment the Python code well. This combined with the general readability of Python code (even by those who don't know the language!) should give an idea of how things are operating.

For tomorrow:

- Implement the processors which produce scales, arpeggios, modes, etc.
- Modify the signalProcessorKey() function to also choose a major or minor key.
- Produce variations of the signalCombinator() function to combine signals in more interesting ways (eg: mode, or weighted mean)

For the rest of the week:

- Plug the Mapping algorithm into the PlayMusic class (this class has been successfully completed, and will already play music).
- Expand implementation to also process the income statement and cash flow in parallel with the balance sheet.
- Implement variations on processor interaction as described in the design document.

Objectives Achieved: Balance Sheet can be parsed. Roughly 50% of basic Mapping Implementation completed.

Wednesday 30th April

I'd charged myself with the task today of being able to use the signals to generate musical sequences. 'Processors' were added to produce scales, arpeggios and broken chords. The starting note and the direction of notes (ascending or descending) are derived from the signal's relative position to a reference signal (usually the mean of all signals from an account sheet). The type of musical sequence for a signal is also derived from its relative position to the reference signal.

Of course, this in its self presents a problem. We will have several different musical sequences, all playing in different (and most likely clashing) keys. Therefore, it was necessary to devise a way to shift all of these sequences into an overall key signature.

The way I did this was to produce an octave template. This is either the 7 notes of 'c' major or 'c' minor scale, minus the final 'c' (so, we have 'cdefgab' for c major). This

octave template is then shifted into a desired key signature. For example, a shift of 2 will put it in 'd' major, using a wrap-around to ensure the number of notes stays the same.

next up, we produce what I call a 'key map'. This is a set of all playable notes, in a key generated from the shifted octave template above.

With this key map at hand, we enforce that all notes in any musical sequence must only use the notes in the map. If any notes clash with the map, they are shifted up. This way, all musical sequences will be in the specified key.

Additionally, the signalProcessorKey() function now chooses a major or minor key.

For tomorrow:

- Link code up to the PlayMusic class.
- Listen to sounds produced and form an opinion on the implementation's progress.
- Tweak settings within the code to see how it effects the sound.

For Friday and the weekend:

- Expand account inputs and processor interaction.
- Perform preliminary evaluation.

Objectives Achieved: Most of Mapping Implementation completed, pending review.

Thursday 1st May

Today was almost a disaster. An obscure bug in Jython meant that certain Java classes crashed when called directly, where as these same classes would work fine when called from another Java class.

What this meant for the project is that the Java class which plays the music doesn't work when called from the Mapping class in Jython. Therefore, today was spent exploring a temporary workaround for this problem.

The result, is that the Jython Mapping class writes the musical output to a CSV file, which is then read in from the Java PlayMusic class. This sadly breaks the seamless integration I was planning, and has meant I lost a day fixing this issue. On the more positive side, I had a day's buffer set aside this week for just such an unforeseen situation, and things are now working again.

Objectives Achieved: Damage control.

Friday 2nd May

As my classes are starting to reach the point where tracking them mentally is difficult, I'm going to list the available functions in the Mapping class:

prepareAccounts():returns accounts - Reads in two CSV files and returns a list of pairs for each attribute.
generateSignals(accounts):returns signal - Takes in accounts and returns a list of signals for each pair.
signalCombinator(signals):returns signal - Takes in many signals, and returns one signal which is the output of a function on the inputs.

signalProcessorTempo(signal):returns tempo - Takes in a signal, and returns a tempo.
signalProcessorKey(signal, referenceSignal):returns (keyNote, keySig) - Takes in a signal, and returns a key signature.
signalProcessorSequence(signal, referenceSignal):returns musicalSequence - Generates a music sequence from a signal and a reference signal.
signalProcessorScale(signal, signalVariance):returns scale - Generates a scale from a signal and some variance.
signalProcessorArpeggio(signal, signalVariance):returns arpeggio - Generates an arpeggio from a signal and some variance.
signalProcessorBrokenChord(signal, signalVariance):returns brokenChord - Generates a broken chord from a signal and some variance.

createFullScale(octaveTemplate):returns allNotes - Takes a template of 7 notes in a scale, and returns all notes over the entire set of MIDI notes.
shiftKey(octaveTemplate, keyShift):returns shiftedOctave - Shifts the notes in octaveTemplate to a different key specified by keyShift.
getKeyMap(octaveTemplate, keyShift):returns keyMap - Combines shiftKey() and getKeyMap() to produce all notes in a given key.
getKeyName(noteValue):returns noteName - Gives the name of a note from a MIDI value.
mapToKey(musicalSequence, keyMap):returns newMusicalSequence - Coerces musicalSequence into a key given by keyMap.

playMusic(musicalSequences):returns formatCheck - Sends musicalSequences directly to the PlayMusic java class to be played.
dumpToFile(musicalSequences):returns formatCheck - Dumps musicalSequences to a CSV file, which can be read in by the PlayMusic java class.
outputMusic(musicalSequences):returns formatCheck - Chooses from playMusic() and dumptToFile() depending on a setting in the code.

To put into perspective how these functions are used, here is some pseudocode to produce a list of musicalSequences from two account files:

1. Let **accounts** = list of pairs of attributes from two account files
2. Let **signals** = signals generated from **accounts**
3. Let **referenceSignal** = a combination from all of **signals** (usually the mean)
4. Let **musicalSequences** = empty list
5. For each **signal** in **signals**:
    1. create a **musicalSequence** for **signal**
    2. add **musicalSequence** to **musicalSequences**
3. Return **musicalSequences**

And, voila! We have music!

Unable to contain my excitement, I played the music produced for some accounts to various acquaintances sharing the lab with me. The general feedback was that it was possible to tell whether an account was in a healthy state or not. As expected, attributes improving between years play ascending sequences, whilst those attributes that are getting worse play descending sequences. More so, the greater the change in the attribute, the 'wider' the musical sequence. The result is that when all these sequences are combined, the overall movement seems to give a feel of the account.

However, it's not perfect. Here are some issues that still need to be addressed:

- The choice of starting note needs to be perfected. Under what circumstances should the starting note be low, or high?

- The cut-off points which are used to decide what type of musical sequences are used need to be tweaked. For example, how big does the change between years of an attribute need to be before we start using arpeggios instead of scales?
- Timing needs to be applied so that shorter musical sequences are spread out properly.
- All musical sequences are in the same key. But, if there is an attribute in the account which is particularly bad, perhaps it shouldn't have its key adjusted to fit with the rest so that it deliberately clashes and makes the music sound unpleasant.

Objectives Achieved: Working prototype for Mapping Implementation. Music played.

Saturday 3rd May

Regarding the issue of what starting note should be chosen, I considered that when listening to a piece of music, the melody tends to lie in the middle of the audio range. Therefore, it makes sense to order the signals into a list based on their relevance. The most relevant signal plays music in the region of 'middle c', whilst less relevant signals play further a way from this centre point.

The next question which arises is how do we choose the ordering of the signal list? Well, the further a signal deviates from 1.0, the more of a change there has been in that account attribute.

To this end, we have a new function, which returns an encoding giving the order of the signals: **orderSignals(signals):returns orderedSignalEncoding**

So, we arrive at the subjective section. As I sat there listening to the music being produced by the accounts, I wasn't too pleased with the number of unintentional note clashes I was hearing. This is due to the fact that there are five musical sequences playing at the same time.

In order to evaluate the nature of the note clashes, I opened up the CSV file produced by that Mapping class, and cut/pasted all the sequences onto the same line so that they would play sequentially instead of in parallel.

The result was something that sounded very melodic, and the key changes between sequences provided context instead of clashes. I've written a flag into the program which can be altered to choose between sequential and parallel output.

Coming back to the note clashes, I realised that the reason the parallel playing sounded bad was because I'd not called the **mapToKey()** function. Adding this in removed the clashes, but left the music sounding quite bland.

So, as a result of this, the Mapping implementation has two modes we can use:
- **Sequential**: All musical sequences are played one after the other in their own keys.
- **Parallel**: All musical sequences are played at the same time in a global key.

I will get some opinions next week as to which works better in terms of giving an overall context of the account. Likely, the evaluation phase will look at both of these systems, as they both have their merits.

Samples of the accounts described in the 29th April entry can be heard here:
- **Sequential**: http://www.macs.hw.ac.uk/~dd20/financialmusic/files/sequential.mp3
- **Parallel**: http://www.macs.hw.ac.uk/~dd20/financialmusic/files/parallel.mp3

Tomorrow, I hope to put the finishing touches to this implementation.

Objectives Achieved: Fine tuning and improvement of Mapping Implementation.

Sunday 4th May

I'm not even sure I woke up today.

Objectives Achieved: Unknown.

Monday 5th May

I've narrowed down the few final changes that need to be implemented in order to have a completed Mapping implementation:

1. Tempo needs to be derived from the rate of change of account elements.
2. Key needs to be derived from the overall direction of movement (major = account improves, minor = account declines).
3. CSV output file needs an additional line to indicate tempo.
4. The function which shifts notes into a given key needs to be more versatile (I will discuss this shortly)

For point (1), The first question is how to derive the overall amount of change from the account? Recall that when we're deriving signals from the change between two attributes, we notice that if the signal > 1, the attribute is increasing and if the signal < 1, the attribute is decreasing. Therefore, we can conclude that the rate of change is the amount in which the signal deviates from 1.0. If we map this value of 'no change' to a very slow tempo (say, 80bpm) then all we need to do is increase the tempo in line with this deviation. So, a deviation of 0.1 produces a tempo of approx 90bpm. I have also introduced a static variable called **TEMPO_SPEEDUP** which is used to alter the ratio of signal deviation versus tempo.

For point (2), we simply check whether overall the accounts status is improving or not. If it is, use a major key. If not, use a minor key. If it's roughly staying the same, we use an ambiguous pentatonic scale which has elements of both major and minor keys.

Point (3) is a simple matter of appending the tempo to the first line of the CSV file, and ensuring the the **MusicReader** class parses it correctly.

With regards to point (4) above, the reason that I declared music produced by the parallel implementation of the Mapping algorithm to be 'bland' is because the notes are shifted upwards into the correct key. This is not the best approach. Instead, I have re-written the **mapToKey()** function to move keys in such a way that the spread of possible notes is better distributed. This helps reduce the blandness, but does re-introduce a limited risk of clashing notes.

For example, consider the chromatic scale: [50, 51, 52, 53, 54, 55, 56, 57, 58, 59].
the old function would map this into a c major scale by producing: [50, 52, 52, 53, 55, 55, 57, 57, 59, 59].
The improved function produces the following note sequence: [50, 50, 52, 53, 53, 55, 55, 57, 57, 59].

Looking at the implementation as a whole, I feel that there isn't a particularly large scope for emergent behaviour in these algorithms. I'm hoping that the Genome approach will fare better in this area.

The next task to hand is to decide which functions from the Mapping implementation are going to be reused in the other implementation, and put them into their own class library which can be imported. These functions are those which read in the account CSV files, generate signals and write the music.

Before I begin tomorrow on the Genome approach, I wanted to have a quick try at designing and implementing a simple L-System approach to complement the other two. To this end, I have used the formal L-System definition (V, S, w, P) to implement a framework in Jython consisting of three functions. **getLSystem(V, S, w, P)** validates the parameters given to it to ensure they define a complete L-System, and then it returns an L-System instance. **runLSystem(lSystem)** takes in one of these instances, and applies *P* against *w* returning a complete L-System instance with a modified *w*. By repeatedly calling this function, the rules of the L-System can be applied many times. Finally, **getAxiom(lSystem)** simply returns *w* from an L-System instance. It is intended to be used to extract the final result.

In designing the grammar, I'm currently being inspired by this web page: http://www.pawfal.org/index.php?page=LsystemComposition

Here is an idea for a grammar:

V (variables) = [A, B, ...]
S (constants) = [s, r, u, d] where s = sustain, r = rest, u = pitch up and d = pitch down.

By the rules of an L-System, applying *P* to *w* will cause all instances of *V* to be replaced, but instances of *S* will remain unchanged. Likewise, when we turn the output into music, members of *V* will be ignored (consider them like Junk DNA).

Now, the rules themselves in *P* will be generated from the accounts. Similar to the way that the *Mapping* implementation produced signals from account attributes, this time we will instead produce rules.

(note: I accept the fact that I am probably mad for attempting to implement a third approach whilst the second still needs to be done. I believe that the L-System approach can be implemented extremely quickly, and is really worth pursuing!)

In terms of the time scale, I'm giving myself until Tuesday week (**8 days from tomorrow**) to have all implementation completed. Although a couple of days longer than I'd planned for the implementation period, this will give me enough leeway should I run into unexpected problems implementing the Genome approach. Conversely, if all goes well, this time limit will prevent me from getting too carried away trying to implement these ideas (which, knowing myself as I do, is *very* likely to happen).

Anyway, here's the plan for tomorrow:

- Begin implementation of Genome approach.
- As a second priority, I will be considering ways of generating L-System rules from accounts.

Objectives Achieved: **Fully completed Mapping Implementation**. Shared functions relocated to shared library. L-System approach considered. L-System framework implemented in Jython.

Tuesday 6th May

To begin with, I've begun the process of gathering testers to evaluate the prototypes. I

have about a week to do this, but I'm not expecting this to be a problem as some people have already expressed interest, mainly because of the bizarre nature of this project.

The first revelation of the day was that Jython is based on an older version of Python (Python is currently at v2.5.1, where as Jython is at v2.2.1). This means the lack of the incredibly useful **set()** operator. So, I'm going to have to implement workarounds based on lists, which won't be quite as elegant as I'd hoped.

Moving on from this, recall that from the second deliverable I had produced a collection of prototype functions for the Genome Approach. These functions don't really do much more than represent the ideas presented in the write-up, but they do provide a good foundation for beginning the implementation of the Genome Approach.

Reading back over the design document for the Genome Approach, two things leap out at me. Firstly, is that it needs to be implemented in such away that some kind of machine learning can be implemented. Secondly, is that the word tables that are used to assign genes are really a series of if/then statements. If we combine these two observations, we find that if we come up with a decent encoding, we may be able to implement ideas from **Alex Freitas**' Paper: *A Survey of Evolutionary Algorithms for Data Mining and Knowledge Discovery*.

In this instance, we need our rules to take some account information in its raw form, and use this to predict an appropriate financial buzz phrase (such as 'soar'). This buzzword represents one expression of the gene, with an alternate expression being derived from the word table (could be 'ascending scale'). In this sense, we are using accounts to derive the phenotype of a gene.

Thinking ahead to the testing, I think a suitable approach would be as follows:

1. Choose a list of buzz-words to represent accounts.
2. For each account have an expert assign any number of these buzz-words to each account.
3. Have the testers do the same by listening to music generated from these accounts.
4. See how the choice of words from the testers compares to the choice of words from the expert.

In other news, Facebook is fantastic for drumming up testing volunteers!

Objectives Achieved: Initial set-up of Genome Implementation.

Wednesday 7th May

I currently have 11 testers lined up for next week. As some of them are "off-site" (or, to put it another way, on the other side of the country/world/universe/multiverse), I'm going to set up the test on a web page with embedded sound files and a way of submitting responses to an email address.

Coming back to the Genome Implementation, I have restructured the gene definition using the following dictionary format:

{ GeneID: [FinancialKeyword, [MusicKeyword1, MusicKeyword2]] } Where GeneID is a Godel number derived from the three keywords.

Next, we need to set up a way that ensures only valid genes are used. For example, the gene:

{ 108: ['Plunge', ['Minor', 'Scale']] }

is valid (although may not be a good gene to represent account mapping, but I'll come to that in a moment). However, the gene:

{ 300: ['Plunge', ['Ascending', 'Chord']] }

is invalid, as we can't have an ascending chord (also, using an ascending sequence for a 'plunge' is probably not a good idea! Again, we'll get to this issue later on).

So, what we need is a way of taking the set of all genes *GN* and filtering out the genes with invalid musical sequences to produce a set of valid genes *GN_VALID*. This is done by defining which musical elements can be paired up. For example, to specify what can be paired with 'Descending', we define a pair: ('Descending', ['Scale', 'Crescendo']).

We create a list of these pairs and name it *W_VALID*. This is then passed to a function along with the full genome: **getValidGenes(GN, W_VALID):returns GN_VALID**.

It's at this point that I need to deviate slightly from the design document. As we have a set of genes which *(a)* might be present in the genome of a given account and *(b)* of these, some will produce music which will represent the account better than others.

Therefore, as we have an optimisation problem it seems to make sense to implement an evolutionary algorithm to determine which genes are best for representing accounts. This will involve supervised machine learning, with myself (and hopefully some others) evaluating several musical sequences per account and helping the software learn which genes are best.

Today's work has been concerned with expanding an underdeveloped idea from the last deliverable, and in my head a lot more ideas have been put together. Tomorrow I expect to move on from writing code which represents the idea, to code which actually does something very cool.

Things to do tomorrow:

- Optimise the gene structure slightly (more on this in the next entry).
- Create an encoding using an enumeration.
- Investigate a strategy for evolution.
- Investigate a strategy for supervised learning.

Things for the rest of the week:

- Implement evolution and machine learning.
- Gather some volunteers for the supervised learning.

Objectives Achieved: Approach further developed. Implementation progresses.

Thursday 8th May

First up, some minor changes to the overall framework so CSV files can be specified when running the Mapping class. I fixed a number of bugs in the Mapping implementation, and it works a little smoother because of this.

A meeting with my supervisor today suggested I reconsider the scope of the Genome implementation. More on this later.

Modifications made to the Genome approach involved separating musical elements into

their own category, so we now have:

{ geneID: ['Financial Keyword', ['Direction', 'Key', 'SequenceType']] }

So, we could have: ['Plunge', ['Ascending', 'Minor', 'Crescendo']] in the dictionary, or ['Boom', ['NULL', 'Minor', 'Chord']]. Notice that I've added NULL types to be used where a musical attribute is not relevant. The rules to deduce valid genes are slightly more involved because of this, but the principle remains the same.

Notice, that each gene can be read as a logical statement. Take for example:

['Plunge', ['Ascending', 'Minor', 'Crescendo']]

This can be read as "Ascending Minor Crescendo IMPLIES Plunge". Using a logical rule (I forget what its called!) we can rearrange this as "IF Plunge THEN Ascending Minor Crescendo".

As we have used NULL values to avoid absurd situations such as "Minor Major Scale", we can look to ideas in the Freitas paper. Even though Freitas' ideas were intended for classification problems, I wonder how these can be used towards an optimisation problem such as this.

The idea is, that we generate a random set of genes for account, and evaluate this set's fitness by listening to the music it produces, and comparing it to the account its self. Depending on the evaluation, we implement a hill climbing algorithm to produce a new gene which may better approximate the account.

Over time, we build up a picture of which genes go with certain types of accounts, and our predictions improve.

Of course, the training its self is a monumental task involving lots of experts to evaluate lots of accounts directly, and lots of testers to listen to the music. But, I believe the idea can be developed into a neat prototype.

Other work done today involved creating an encoding for the genes, and functions to perform *single-random-gene new-allele mutations* as well as *m-random-gene new-allele mutations* upon our encoding. I've tested all these functions to my satisfaction. Genes also record weights, which might be an alternative way of tracking fitness.

Moving on to the L-System approach, my supervisor suggested that an alternative to generating the rules from an account would be to generate the start axiom from the account, and then use a fixed set of rules. He also pointed out that it could be tough to make a musical sequence of notes, rather than just a random sequence of notes.

Here is a current list of things to be completed before the write-up:

- Investigate Freitas approach towards training the Genome Approach.
- Use a few volunteers to perform an experimental evaluation of the Genome Approach Prototype.
- Continue development of L-System approach as quickly as possible.
- Carry out full evaluation of Mapping Approach (and L-System Approach if ready) using testers.
- Connect approaches to GUI (if time).
- If one or more evaluations are successful, create a web applet (if time).

Objectives Achieved: Further development of Genome Approach.

Sunday 11th May

After a couple of days off, I'm back to work. Today, I pushed forward with the L-System Implementation a bit. To complement the formal framework, I also added a parser which turns the output axiom into a sequence of MIDI notes, which can be played using the PlayMusic class as per usual. The parser has its own set of rules to interpret the axiom, and supports the use of a stack via the '[' and ']' symbols. To this end, these symbols are reserved in the L-System framework, and cannot be specified as part of *V* or *S* (although they are automatically added to *S* by the function which returns an L-System).

Objectives Achieved: Music produced by L-System can be played.

Monday 12th May

More work on the L-System Approach. An axiom produces two strings of MIDI notes. The first is the main melody (although, how 'melodic' this is remains to be seen). The second is an optional harmony, which is played either 3 or 4 notes above the current note in the melody (3 notes = minor, 4 notes = major). The harmony can be turned on or off by the use of items in the axiom grammar.

I've kept things simple for the time being, so that we have to worry about just producing a single melody sequence of notes from an account. The second harmony sequence is derived from the melody sequence, so we don't have to be overtly concerned with it. If I can get this working, I'd like to add some chords as well.

So far, we have the following grammar which can be played (all other grammar is ignored):
'u' - Go up in pitch
'd' - Go down in pitch
's' - Stay on same note
'r' - Don't play anything
'.' - Key up
',' - Key down
'+' - Turn on harmony
'-' - Turn off harmony
(note that the stack operators '[' and ']' are implicitly built into my L-System model.)

The next step is to use the L-System to produce some music from accounts. To do this, I'm going to use the signals derived from the Mapping implementation as a basis for generating a starting axiom, to which fixed rules will be applied.

Consider that we can grade signals from A to F. We can then define set *V* as ['A', 'B', 'C', 'D', 'E', 'F']. Rewriting rules are then specified to rewrite these variables. Therefore, the starting axiom for an account based on the 'quality' of its various attributes might be ['B', 'C', 'E', 'E', 'C'].

The way that a note is incremented or decremented is via two functions **pitchUp(note, keyMap)** and **pitchDown(note, keyMap)**. The keyMap passed to the function is used to work out where the next note in the current key lies. This way, we can switch key in the middle of a note sequence without causing clashes. Also, the key can shift independently of the individual note sequences, providing musical coherence.

Add to this two self explanatory functions: **keyShiftUp(keyMap, shiftDistance)** and **keyShiftDown(keyMap, shiftDistance)**.

Note also that I have stuck with the convention of using upper case characters for members of *V* and lower case characters for members of *S*. The reason for this is

because Python distinguishes between character cases, so there is less chance of an accidental duplication of a symbol (although my L-System framework does check for these situations).

Tomorrow's work will involve experimenting with rule sets to find something that re-writes the starting axiom appropriatly.

Objectives Achieved: Further progress with L-System.

Tuesday 13th May

Today I'll be experimenting with re-writing rules. In terms of the L-System, when I use the term "reduction" I'm referring to the removal of members of *S* from *w*. A fully reduced axiom will therefore have no occurrences of *S*.

In the results section for each set of rules, the first line is the starting axiom, and then each further line demonstrates an application of the rules to the axiom.

Rule Set #1
ruleA = ('A', 'uuuB')
ruleB = ('B', 'uuC')
ruleC = ('C', 'u')
ruleD = ('D', 'd')
ruleE = ('E', 'ddD')
ruleF = ('F', 'dddE')

Results #1
['B', 'C', 'E', 'E', 'C']
['u', 'u', 'C', 'u', 'd', 'd', 'D', 'd', 'd', 'D', 'u']
['u', 'u', 'u', 'u', 'd', 'd', 'd', 'd', 'd', 'd', 'u']

Evaluation #1
These rules result in a fully reduced term after just two applications. If there were 'A's and 'F's in the term, it would probably take three iterations to fully reduce the term. Music moves up and down within a small range of notes, and sounds uninteresting. Music is short.

Rule Set #2
ruleA = ('A', '[uuuB]')
ruleB = ('B', '[uuC]')
ruleC = ('C', 'u')
ruleD = ('D', 'd')
ruleE = ('E', '[ddD]')
ruleF = ('F', '[dddE]')

Results #2
['B', 'C', 'E', 'E', 'C']
['[', 'u', 'u', 'C', ']', 'u', '[', 'd', 'd', 'D', ']', '[', 'd', 'd', 'D', ']', 'u']
['[', 'u', 'u', 'u', ']', 'u', '[', 'd', 'd', 'd', ']', '[', 'd', 'd', 'd', ']', 'u']

Evaluation #2
Music is spread wider thanks to the use of a stack memory, and even resolves. Sounds slightly more interesting than #1.

Rule Set #3

ruleA = ('A', '[..uuuC]')
ruleB = ('B', '[uuA]')
ruleC = ('C', 'Bu')
ruleD = ('D', 'Ed')
ruleE = ('E', '[ddF]')
ruleF = ('F', '[,,dddD]')

Evaluation #3
Longer sequences of scales with jumps across the spectrum to pick up new scales. The key changes are hard to detect from the notes alone, which appear seemingly random. At this stage I'd like to add a chord underneath the melody which will play in the current key. I hope this will give the melody more context than it currently has.

What happened next was a modification to the system so that a chord plays underneath the melody. The chord is the triad of the current key, and remains held down until a key change, when the chord shifts into a new key.

However, although all notes are coerced into the correct key, things still sounded quite discordant. Therefore, when the key shifts, the shift is added onto each note. This makes the music nice and tuneful.

Still to Do
L-System: Set initial major or minor key depending on the account's overall state.
L-System: Expand grammar so key can switch between major & minor.
L-System: Set tempo in similar fashion to Mapping Implementation.
L-System: Expand grammar for note shifts greater than 1 (ie, repeatedly call **pitchUp()** and **pitchDown()**, etc.)
L-System: Set limit on music length, and only apply L-System the number of times needed to get this length.
L-System: Musical sequences should somehow resolve.
L-System: Devise and evaluate many more rule sets (of-course, this is the fun part ;-)
Main Framework: Add a dummy signal generator, so that signals can be generated without accounts to see how the music changes.

At the stage I'm at now, I'm pretty confident that testing can begin soon on the Mapping and L-System implementations. I'm aiming to begin in two days from now, which gives me tomorrow to tweak both of these implementations to my satisfaction.

With regards to the Genome implementation, I will work on this at the same time as the testing is going off. The objective here is to have a working prototype which can be trained, although not on a scale where the training would actually be effective. I will present this implementation in the report as something which can be expanded upon in a future project.

The testing strategy will consist of the following (with numbers to be decided)

x accounts
for each account:
    Mapping sequential music
    Mapping parallel music
    y number of L-System 6 grade music
    z number of L-System 10 grade music

All this music will be shuffled randomly for each listener. Numbers are to be decided, and I will explain the testing process in greater detail very soon. I'm reminded to research issues of music fatigue, as testers may have to listen to large amounts of music (although each piece should be quite short).

Objectives Achieved: L-System implementation completed.

Wednesday 14th May

One of the considerations for formulating L-System rule sets is that they should be set up such that chord/key changes only happen at regularly spaced intervals. This is to give the impression of the song having a proper time signature, such as four beats per bar. This will help the music generated to actually sound like music.

A change I wanted to make to the framework was to have a number of pre-set instrument schemes. This means that the file format that the music its self is written to needs modification.

I have implemented the ability to change keys from major to minor and vice-versa within the L-System, but this change must occur with the whole chord changing key. So, we add to the grammar 'j' for a change to a major key, and 'n' for a change to a minor key.

At this stage, I have so many settings scattered over the various modules, I'm going to gather them all together in a file called **settings.py**. This way, all settings can be tweaked from one place.

Additionally, the ability to set the approximate limit of a musical sequence has been added. This works by repeatedly running the L-System until the length of the axiom goes over the limit.

Finally, a 10-grade system has also been added to complement the current 6 grades.

(Note to self: Maybe add some basic percussion into the L-System implementation? It won't take long)

For my own reference, here is a list of some of the rules I've been experimenting with:

```
sixGradeRuleA = ( 'A', '[usus..j/r/rE]' )
sixGradeRuleB = ( 'B', '[[usus]..BC]' )
sixGradeRuleC = ( 'C', 'udus' )
sixGradeRuleD = ( 'D', 'dsds' )
sixGradeRuleE = ( 'E', '[[dsds],,ED]' )
sixGradeRuleF = ( 'F', '[dsds,,n_r_rE]' )


sixGradeRuleA = ( 'A', '.j/[/u/]BCA' )
sixGradeRuleB = ( 'B', '[../d/u]CB' )
sixGradeRuleC = ( 'C', 'uud/C' )
sixGradeRuleD = ( 'D', 'ddu_D' )
sixGradeRuleE = ( 'E', '[,,_u_d]DE' )
sixGradeRuleF = ( 'F', ',n_[_d_]EDF' )


sixGradeRuleA = ( 'A', '[,,js///]BCA' )
sixGradeRuleB = ( 'B', '[,,,,///_]B' )
sixGradeRuleC = ( 'C', '/_s/C' )
sixGradeRuleD = ( 'D', '_/s/C' )
sixGradeRuleE = ( 'E', '[...___/]E' )
sixGradeRuleF = ( 'F', '[..ns___]EDF' )
```

```
sixGradeRuleA = ( 'A', '..uuuuB' )
sixGradeRuleB = ( 'B', '[..suud]D' )
sixGradeRuleC = ( 'C', 'uuu_uuu_A' )
sixGradeRuleD = ( 'D', 'ddd/ddd/F' )
sixGradeRuleE = ( 'E', '[,,sddu]C' )
sixGradeRuleF = ( 'F', ',,ddddE' )
```

At this point, I began the write-up, and so further ideas were documented there as they occurred.